차원 축소, 군집화

DNA 2조

이지수, 한지민, 황호진

차원축소

: 다차원 데이터 세트의 차원을 축소해 새로운 차원의 데이터 세트를 생성하는 것

사용 이유:

1. 차원이 증가할수록 희소한 구조 형성
   -> 예측 신뢰도 하락

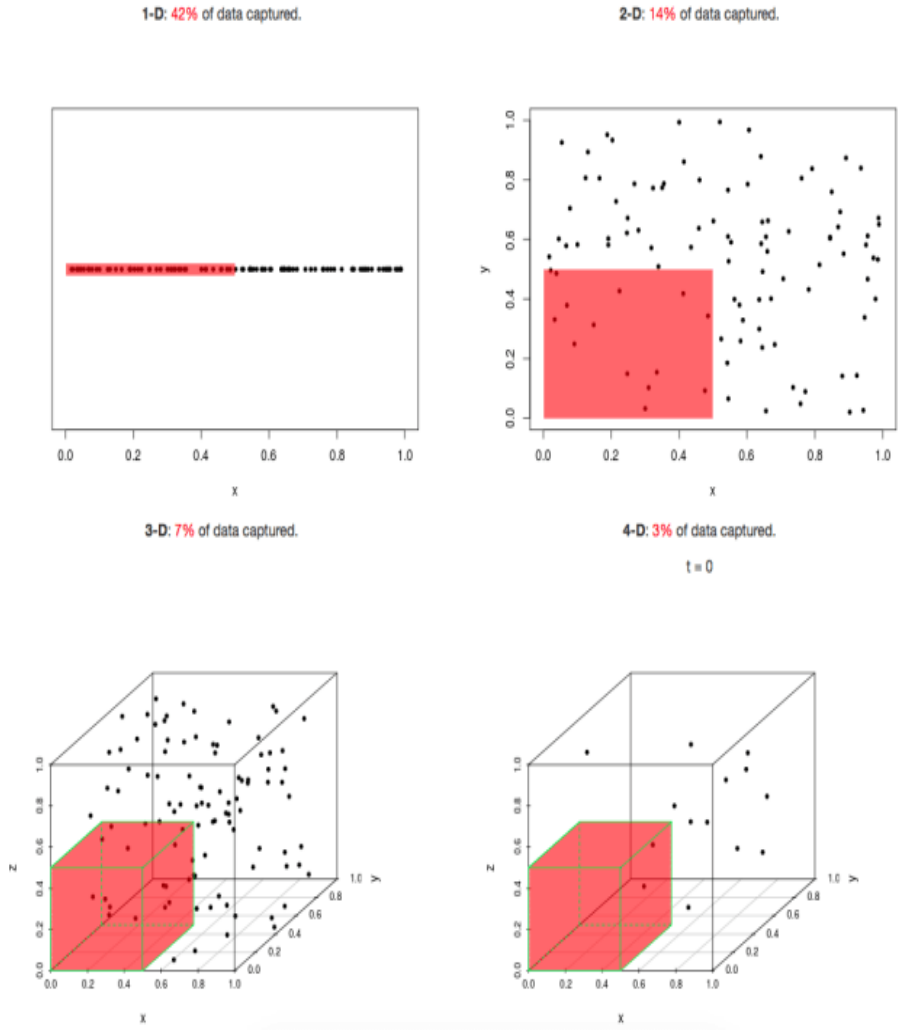
2. 피처가 많으면 개별 피처 간에 상관관계가 높을 가능성이 큼
   -> 선형모델에서 다중 공선성 문제로 모델의 예측 성능이 저하

3. 더 직관적으로 데이터 해석 가능

4. 시각적으로 데이터를 압축해서 표현가능

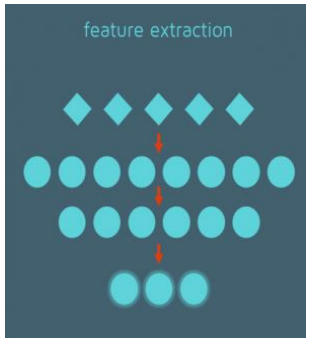
5. 학습 데이터 크기 감소
   -> 학습에 필요한 처리 능력 줄일 수 있다.

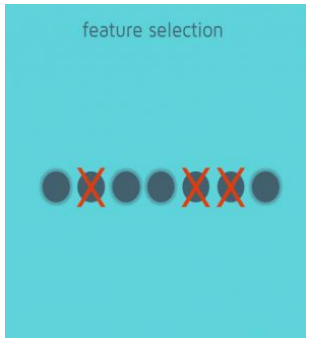
종류 : 1. PCA 2. LDA 3. SVD 4. NMF



★ 차원이 증가 할수록 데이터 포인트 간의 거리가 멀어지는 구조를 가짐.

차원 축소 분류





1. 피처 추출 (feature extraction)

기존 피처를 저차원의 중요 피처로 압축해서 추출

기존의 피처와는 전혀 다른 값

단순압축이 아닌 다른 공간으로 매칭하는 것

2. 피처 선택 (feature selection)

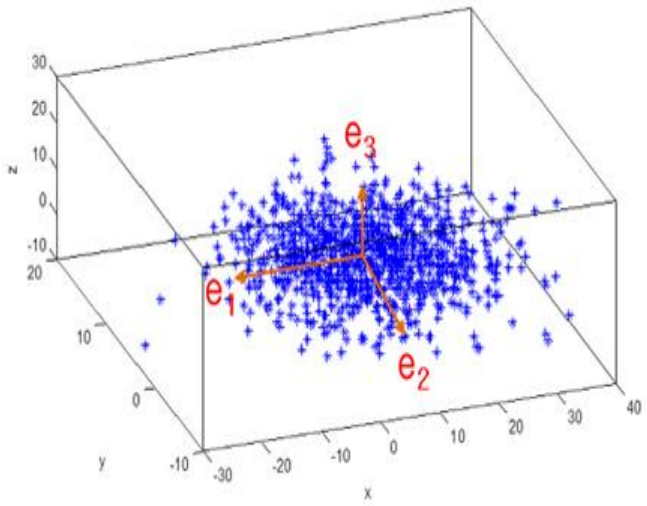
특정 피처에 종속성이 강한 불필요한 피처 제거

데이터의 특징을 잘 나타내는 주요 피처만 선택

1. PCA (Principal Component Analysis)

: 여러 변수 간에 존재하는 상관관계를 이용해 이를 대표하는 주성분을 추출해 차원을 축소하는 기법

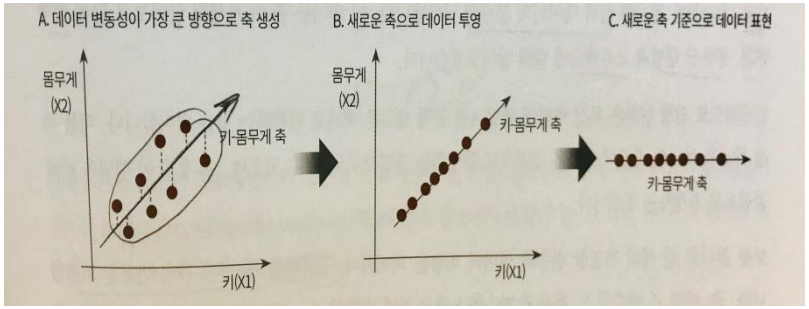
- 주성분: 가장 높은 분산을 가지는 데이터의 축

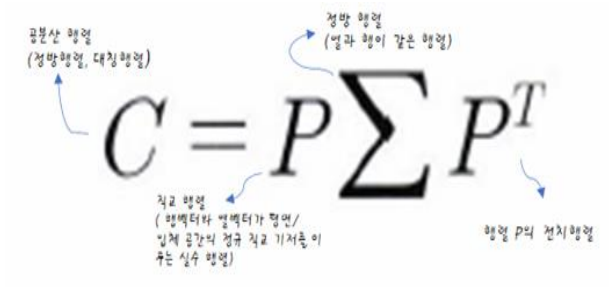


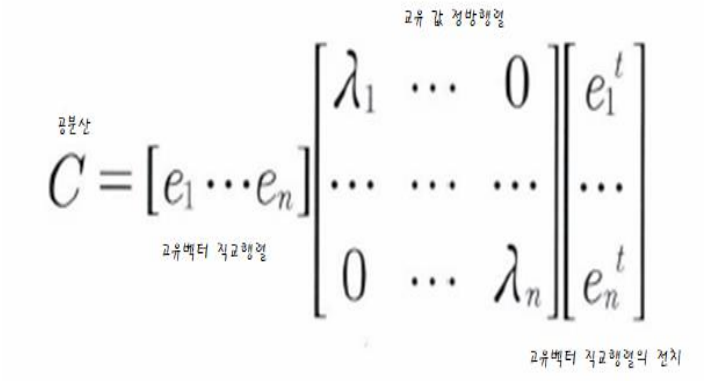
첫번째 벡터 축 : e1 데이터 변동성이 가장 큰 방향
두번째 벡터 축 : e2 / e1에 직각이 되는 직교 벡터
세번째 벡터 축 : e3 / e2에 직각이 되는 직교 벡터

… 벡터 축의 개수만큼의 차원으로 원본데이터가 차원 축소

★ 입력데이터의 공분산 행렬을 고유 값 분해하고 구한 고유벡터에 입력 데이터를 선형 변환하는 것







PCA step

1. 입력 데이터 세트의 공분산 행렬 생성

2. 공분산 행렬의 고유 벡터와 고유 값을 계산

3. 고유 값이 가장 큰 순서로 PCA 변환 차수 만큼 고유 벡터를 추출

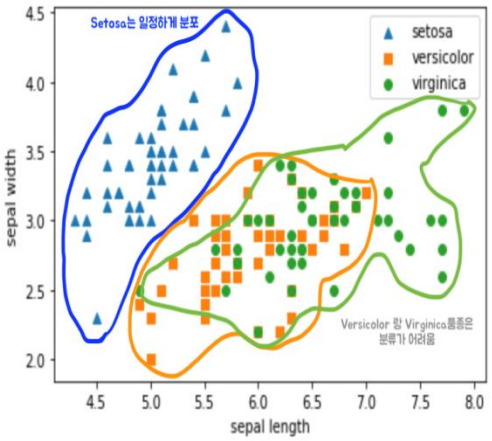
4. 새롭게 입력된 데이터를 반환

붓꽃 데이터 차원 압축

차원 축소 이전 (차원 축소 x)

	sepal_length 꽃받침 길이	sepal_width 꽃받침 넓이	petal_length 꽃잎 길이	petal_width 꽃잎 길이	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

- irisDF['target']=iris.target 사용
- 붓꽃데이터 세트는 4개의 속성으로 이루어짐



> sepal_length를 X축, sepal_width를 Y축으로 시각화 한 품종 데이터 분포

> versicolor, virginica 이 잘 분류되지 않음

압축 과정

Step1) 압축하기 위해 개별 속성 스케일링

- **개별 속성 스케일링**
  > PCA는 여러 속성의 값을 연산하므로 속성의 스케일에 영향을 받는
- StandardScaler().fit_transform(irisDF.iloc[:,:-1]) 사용

Step2) In [32]:

```
from sklearn.decomposition import PCA
pca=PCA(n_components=2)

#FIT() 과 TRANSFORM()을 호출해 PCA 변환데이터 변환
pca.fit(iris_scaled)
iris_pca=pca.transform(iris_scaled)
print(iris_pca.shape)
```

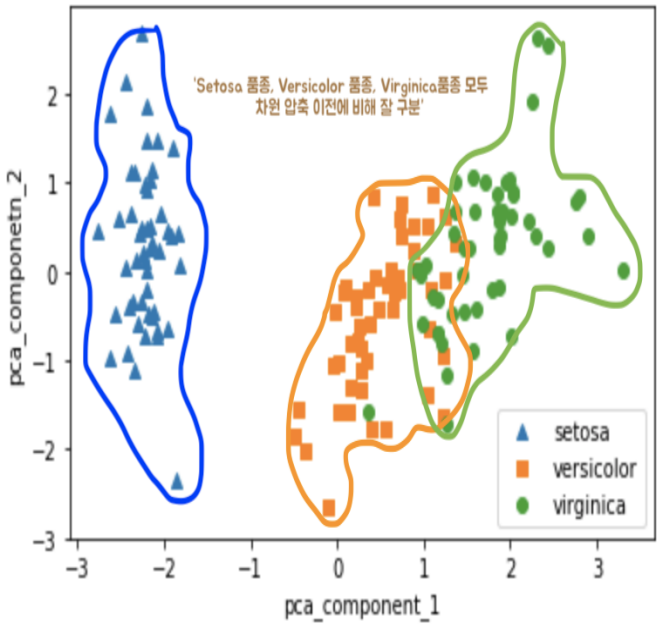
```
(150, 2)
```

PCA는 생성 파라미터로 n_components를 입력
n_components는 PCA로 변환할 차원의 수를 의미

붓꽃 데이터 차원 압축 차원 축소 결과

	pca_component_1	pca_component_2	target
0	-2.264703	0.480027	0
1	-2.080961	-0.674134	0
2	-2.364229	-0.341908	0

> 4개의 속성에서 2개의 속성으로 축소됨



> Setosa, versicolor, virginica 모두 차원 압축 이전에 비해 잘 구분됨.

RandomForestClassifier을 이용하여 교차 검증 세트로 정확도 확인

In [41]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

rcf=RandomForestClassifier(random_state=156)
scores=cross_val_score(rcf, iris.data, iris.target, scoring
='accuracy',cv=3)
print('원본 데이터 교차 검증 개별 정확도:', scores)
print('원본 데이터 평균 정확도:',np.mean(scores))

pca_X=irisDF_pca[['pca_component_1','pca_component_2']]
scores_pca = cross_val_score(rcf, pca_X, iris.target, scorin
g='accuracy', cv=3)
print('PCA변환 데이터 교차 검증 개별 정확도:',scores_pca)
print('PCA변환 데이터 평균 정확도:',np.mean(scores_pca))
```

```
원본 데이터 교차 검증 개별 정확도: [0.98 0.94
0.96]
원본 데이터 평균 정확도: 0.96
PCA변환 데이터 교차 검증 개별 정확도: [0.88 0.
88 0.88]
PCA변환 데이터 평균 정확도: 0.88
```

원본 데이터 대비 10% ↓/ 속성 개수는 50% ↓

➢ **원본 데이터의 특성을 상당부분 유지하고 있음**
> 더 많은 피처를 가진 데이터 세트를 사용하면
예측 성능 저하가 매우 작음 (약 1~2%)

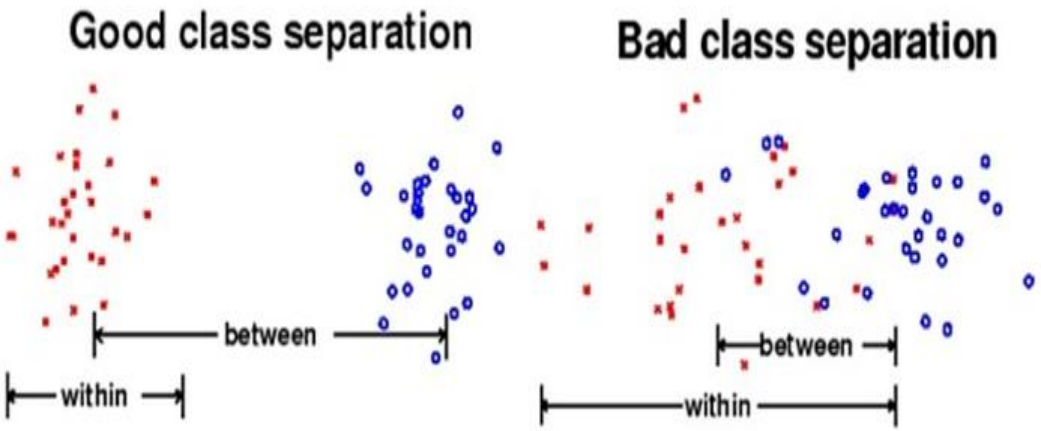
> 얼굴 인식 분야에서 많이 사용

2. LDA (Linear Discriminant Analysis)

: 선형 판별 분석법
: 데이터 세트를 저 차원 공간에 투영해 차원을 축소하는 방법

- PCA와 차이점 :
  분류에서 사용하기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하면서 차원 축소

  즉, 클래스 간 분산과 클래스 내부의 분산을 조절



> 클래스 간 분산은 크게, 클래스 내부의 분산은 최대한 작게

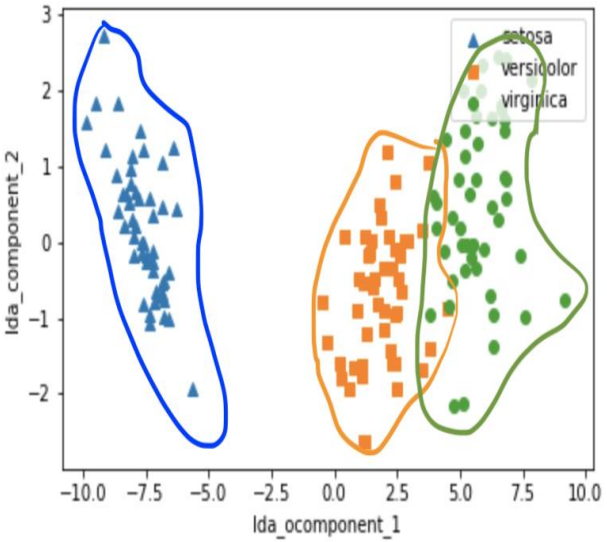
붓꽃 데이터 차원 압축 차원 축소 결과

LinearDscriminantAnalysis 클래스를 사용

```
In [3]: from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
        from sklearn.preprocessing import StandardScaler
        from sklearn.datasets import load_iris

        iris = load_iris()
        iris_scaled = StandardScaler().fit_transform(iris.data)
```

```
In [5]: lda = LinearDiscriminantAnalysis(n_components=2)
        lda.fit(iris_scaled, iris.target)
        iris_lda=lda.transform(iris_scaled)
        print(iris_lda.shape)
```

```
(150, 2)
```



> Setosa, versicolor, virginica 모두 차원 압축 이전에 비해 잘 구분됨

3. SVD (Singular Value Decomposition)

: 행렬 분해 기법 사용
: 넘파이를 통한 SVD 와 사이파이를 사용한 SVD

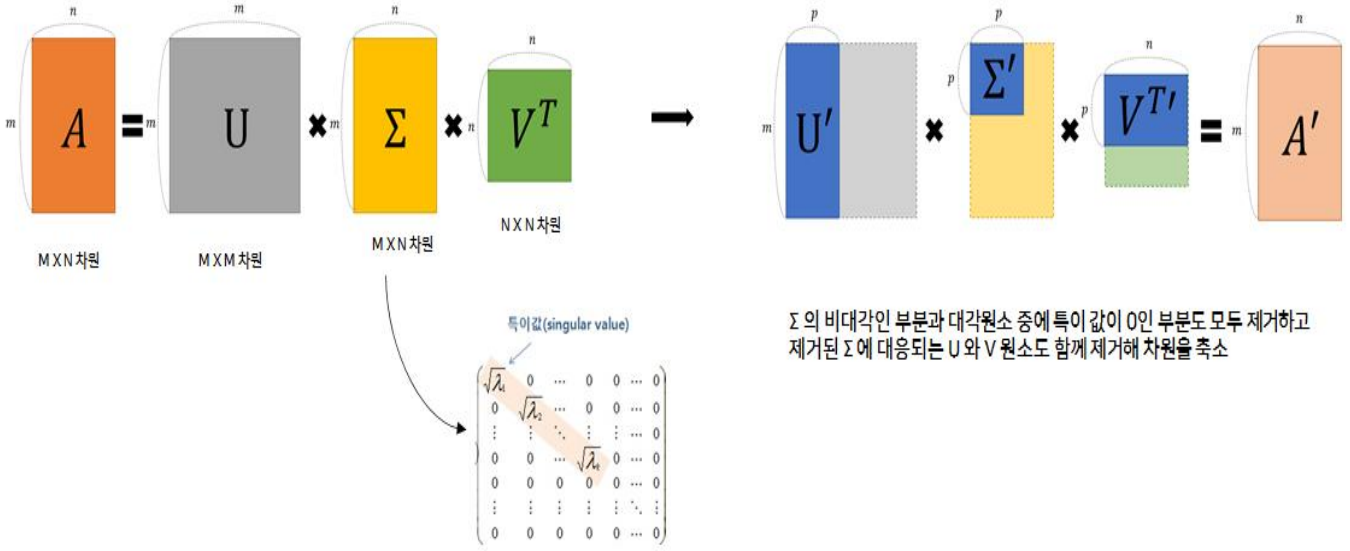
➢ PCA와 공통점과 차이점 :

공통점 : 행렬 분해 기법 사용

차이점 :

PCA : 정방 행렬 (행과 열의 크기가 같은 행렬)에 사용 / 밀집 행렬에 대한 변환만 가능

SVD : 행과 열의 크기가 다른 행렬에도 사용 가능 / 희소 밀집 행렬에 대한 변환



넘파이 VS 사이파이

1. 넘파이

In [6]:

```
import numpy as np
from numpy.linalg import svd

np.random.seed(121)
a = np.random.randn(4,4)
print(np.round(a,3))
```

```
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.014  0.63   1.71  -1.327]
 [ 0.402 -0.191  1.404 -1.969]]
```

In [8]:

```
U , Sigma, Vt = svd(a)
print(U.shape, Sigma.shape, Vt.shape)
print('U matrix: \n', np.round(U,3))
print('Sigma matrix: \n', np.round(Sigma,3))
print('V transpose matrix: \n', np.round(Vt,3))
```

```
(4, 4) (4,) (4, 4)
U matrix:
 [[-0.079 -0.318  0.867  0.376]
 [ 0.383  0.787  0.12   0.469]
 [ 0.656  0.022  0.357 -0.664]
 [ 0.645 -0.529 -0.328  0.444]]
Sigma matrix:
 [3.423 2.023 0.463 0.079]
V transpose matrix:
 [[ 0.041  0.224  0.786 -0.574]
 [-0.2    0.562  0.37   0.712]
 [-0.778  0.395 -0.333 -0.357]
 [-0.593 -0.692  0.366  0.189]]
```

2. Truncated SVD

> 특이값 중 상위 일부 데이터만 추출해 분해하는 방식
> 인위적으로 차원을 작게 만들어 주기 때문에 원본 해열을 정확하게 복구
> 원래 차원의 차수에 가깝게 잘라낼수록 원본 행렬에 더 가깝게 복원 가능
> 모듈 : scipy.sparse.linalg.svds

In [1]:

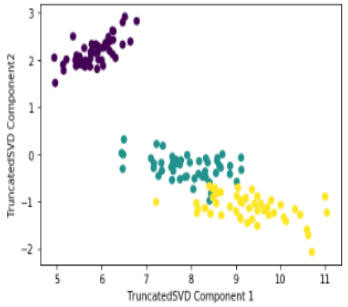
```
import numpy as np
from scipy.sparse.linalg import svds
from scipy.linalg import svd
```

In [10]:

```
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris = load_iris()
iris_ftrs = iris.data
tsvd = TruncatedSVD(n_components = 2)
tsvd.fit(iris_ftrs)
iris_tsvd = tsvd.transform(iris_ftrs)

plt.scatter(x=iris_tsvd[:,0], y=iris_tsvd[:,1], c=iris.tar
get)
plt.xlabel('TruncatedSVD Component 1')
plt.ylabel('TruncatedSVD Component2')
```

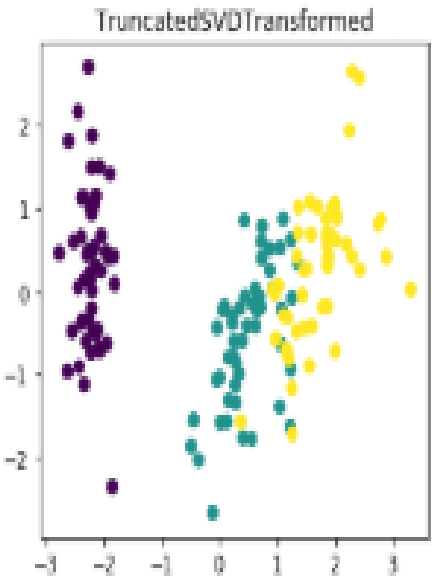
Out[10]:

```
Text(0, 0.5, 'TruncatedSVD Component2')
```

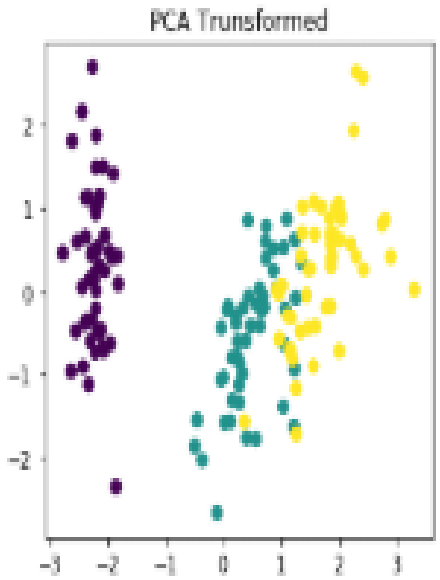


PCA VS SVD

TruncatedSVD 사용



PCA 사용



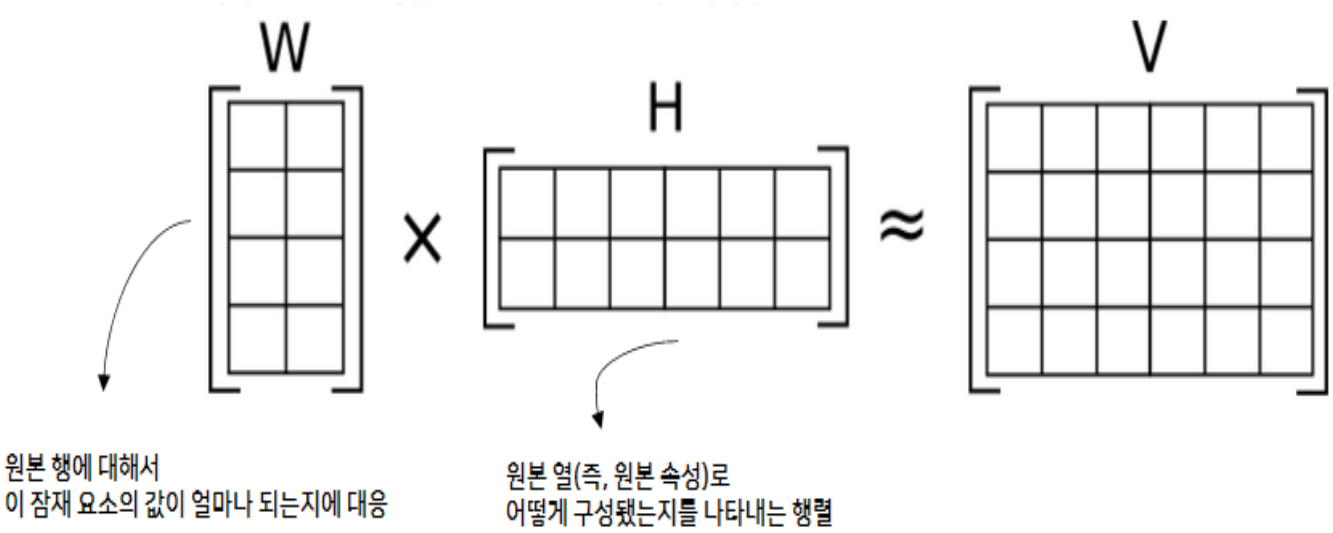
> 2개의 변환이 서로 동일함
> 데이터 세트가 스케일링으로 데이터 중심이 동일해지면 SVD와 PCA는 동일한 변환을 수행

주로 이미지 압축을 통한 패턴 인식, 신호 처리 분야에 사용, 텍스트의 토픽 모델링 기법인 LSA의 기반 알고리즘

EX) https://angeloyeo.github.io/2019/08/01/SVD.html

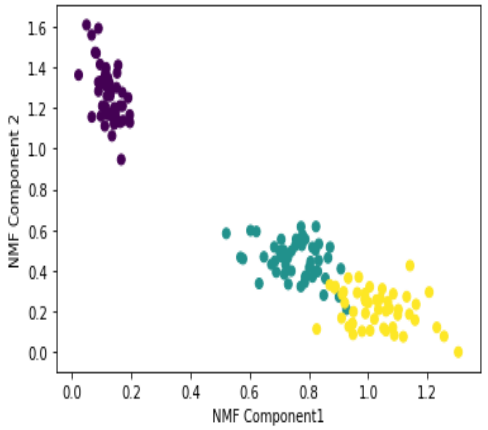
4. NMF (Non_Negative Matrix Factorization)

: 낮은 랭크를 통한 행렬 근사 (Low-Rank Approximation) 방식의 변형



붓꽃 데이터 차원 압축 차원 축소 결과

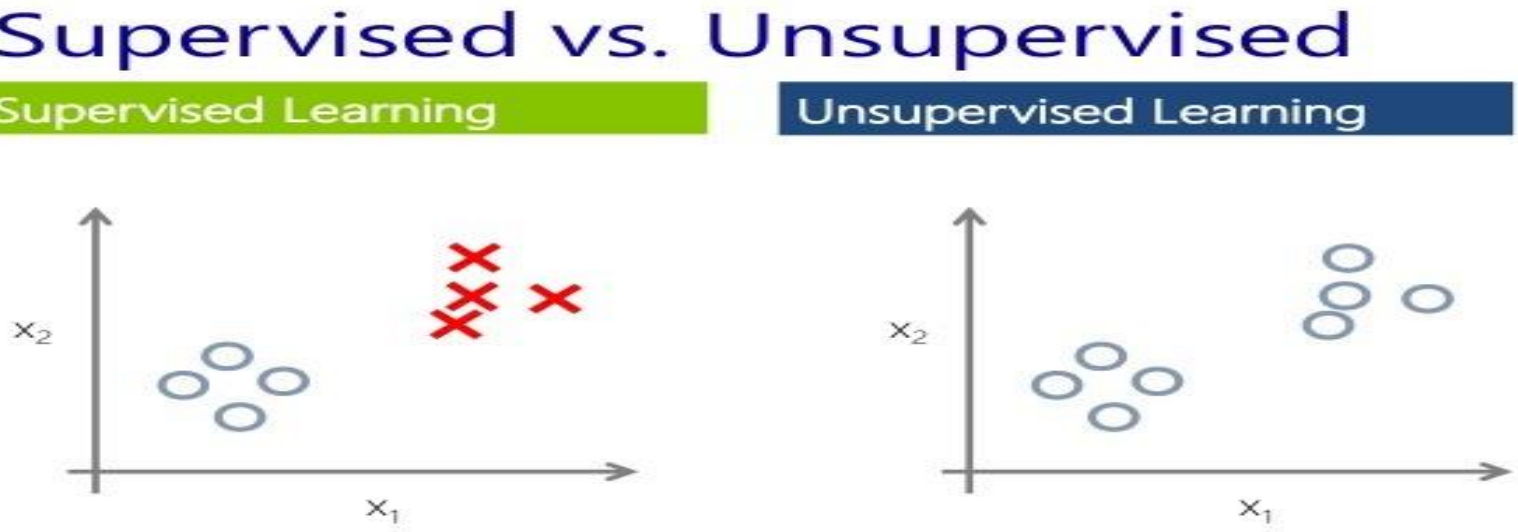
In [16]:
```
from sklearn.decomposition import NMF
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris = load_iris()
iris_ftrs = iris.data
nmf = NMF(n_components = 2)
nmf.fit(iris_ftrs)
iris_nmf = nmf.transform(iris_ftrs)
plt.scatter(x=iris_nmf[:,0], y=iris_nmf[:,1], c=iris.target)
plt.xlabel('NMF Component1')
plt.ylabel('NMF Component 2')
```



이미지 압축을 통한 패턴인식, 텍스트의 토픽 모델링 기법, 문서 유사도 및 크러스터링, 추천(recommendation)영역에서 사

지도 학습 (Supervised Learning)

비지도 학습 (Supervised Learning)



y 값 (target)이 존재

y 값 (target)이 존재하지 않음

- K-means

✓ 대표적인 군집화 방법

  - 장점 : 일반적인 군집화에서 많이 사용, 쉽고 간결함
  - 단점 : 속성의 개수 많을 경우 정확도 하락, 시간이 오래 걸림, 군집 선택의 어려움 발생

✓ 사이킷런 클래스

  - class sklearn.clusters.KMeans()

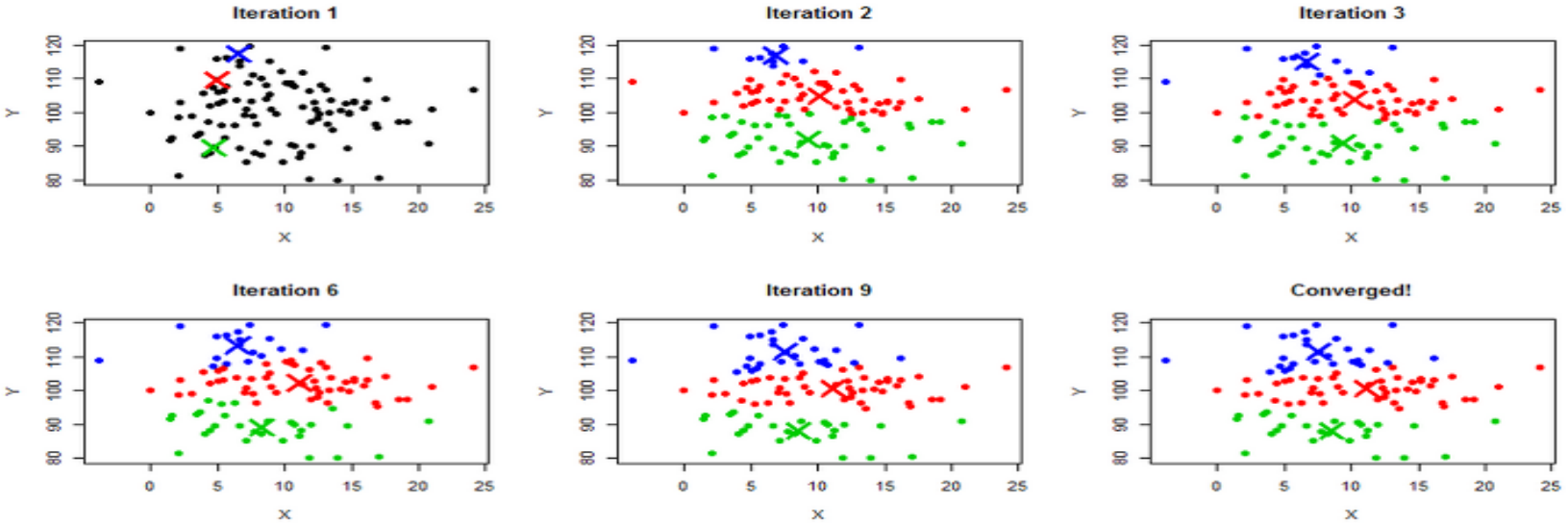
✓ 사이킷런 파라미터

  - n_cluster : 군집화할 개수
  - max_iter : 최대 반복 횟수 (횟수 이전에 모든 데이터의 중심점 이동이 없으면 종료)

✓ 작동 원리

1. k개의 군집 중심점을 설정
2. 각 데이터는 가장 가까운 중심점에 소속
3. 중심점에 할당된 데이터들의 평균 중심으로 중심점 이동
4. 각 데이터는 이동된 중심점 기준으로 가장 가까운 중심점에 소속
5. 다시 중심점에 할당된 데이터들의 평균 중심으로 중심점 이동

* 중심점을 이동했지만 데이터들의 중심점 소속 변경이 없으면 군집화 완료



✓ make_blobs를 활용한 데이터 생성

- 호출 파라미터

✓ n_samples : 생성할 총 데이터의 개수 (default = 100)

✓ n_features : 데이터 피처의 개수

✓ centers : 군집의 개수 (int 값 일 경우), 개별 군집 중심점의 좌표 (ndarray 일 경우)

✓ cluster_std : 생성될 군집 데이터의 표준 편차

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
%matplotlib inline

X, y = make_blobs(n_samples=200, n_features=2, centers=3, cluster_std=0.8, random_state=0)
print(X.shape, y.shape)

# y target 값의 분포를 확인
unique, counts = np.unique(y, return_counts=True)
print(unique,counts)
```

```
(200, 2) (200,)
[0 1 2] [67 67 66]
```

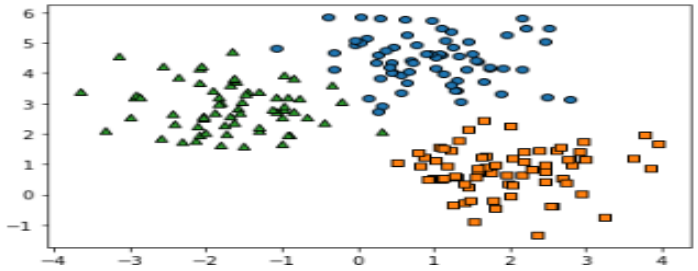
가공의 편리를 위해 DataFrame으로 변경

In [9]:
```
import pandas as pd

clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y
clusterDF.head(3)
```

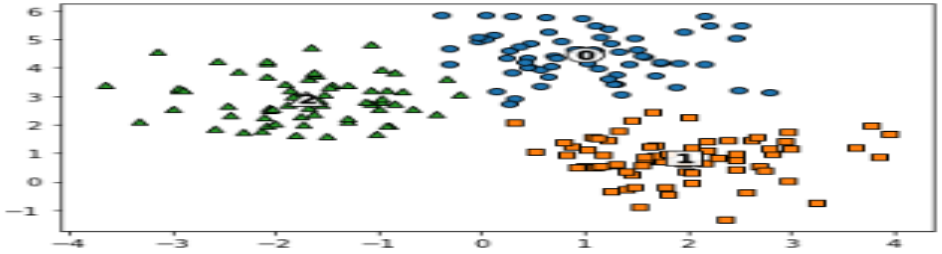
Out[9]:

	ftr1	ftr2	target
0	-1.692427	3.622025	2
1	0.697940	4.428867	0
2	1.100228	4.606317	0

```
target_list = np.unique(y)
# 각 target별 산점도의 마커 값들.
markers=['o', 's', '^', 'P','D','H','x']
# 3개의 cluster 영역으로 구분한 데이터 셋을 생성했으므로 target_list는 [0,1,2]
# target==0, target==1, target==2 로 scatter plot을 marker별로 생성.
for target in target_list:
    target_cluster = clusterDF[clusterDF['target']==target]
    plt.scatter(x=target_cluster['ftr1'], y=target_cluster['ftr2'], edgecolor='k', marker=markers[target] )
plt.show()
```



```
# KMeans 객체를 이용하여 X 데이터를 K-Means 클러스터링 수행
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=200, random_state=0)
cluster_labels = kmeans.fit_predict(X)
clusterDF['kmeans_label']  = cluster_labels

#cluster_centers_ 는 개별 클러스터의 중심 위치 좌표 시각화를 위해 추출
centers = kmeans.cluster_centers_
unique_labels = np.unique(cluster_labels)
markers=['o', 's', '^', 'P','D','H','x']

# 군집된 label 유형별로 iteration 하면서 마커 별로 scatter plot 수행.
for label in unique_labels:
    label_cluster = clusterDF[clusterDF['kmeans_label']==label]
    center_x_y = centers[label]
    plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], edgecolor='k',
                marker=markers[label] )

    # 군집별 중심 위치 좌표 시각화
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=200, color='white',
                alpha=0.9, edgecolor='k', marker=markers[label])
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', edgecolor='k',
                marker='$%d$' % label)

plt.show()
```



✓ 실루엣 분석

- 각 군집간의 거리가 얼마나 효율적으로 분리돼 있는지를 나타낸다.
- 다른 군집과의 거리는 멀수록 좋고 군집 내 거리는 가까울수록 좋다.

✓ 결과 해석

- 값이 1에 가까울수록 근처의 다른 군집과 멀어진다.
- 0으로 갈수록 근처의 군집과 가깝다.
- 음의 값이 나온다면 아예 다른 군집에 갔다고 해석

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

✓ 클러스터 평가

In [6]:
```
from sklearn.preprocessing import scale
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
# 실루엣 분석 metric 값을 구하기 위한 API 추가
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

%matplotlib inline

iris = load_iris()
feature_names = ['sepal_length','sepal_width','petal_length','petal_width']
irisDF = pd.DataFrame(data=iris.data, columns=feature_names)
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300,random_state=0).fit(irisDF)

irisDF['cluster'] = kmeans.labels_

# iris 의 모든 개별 데이터에 실루엣 계수값을 구함.
score_samples = silhouette_samples(iris.data, irisDF['cluster'])
print('silhouette_samples( ) return 값의 shape' , score_samples.shape)

# irisDF에 실루엣 계수 칼럼 추가
irisDF['silhouette_coeff'] = score_samples

# 모든 데이터의 평균 실루엣 계수값을 구함.
average_score = silhouette_score(iris.data, irisDF['cluster'])
print('붓꽃 데이터셋 Silhouette Analysis Score:{0:.3f}'.format(average_score))

irisDF.head(3)
```

```
silhouette_samples( ) return 값의 shape (150,)
붓꽃 데이터셋 Silhouette Analysis Score:0.553
```

Out[6]:

	sepal_length	sepal_width	petal_length	petal_width	cluster	silhouette_coeff
0	5.1	3.5	1.4	0.2	1	0.852955
1	4.9	3.0	1.4	0.2	1	0.815495
2	4.7	3.2	1.3	0.2	1	0.829315

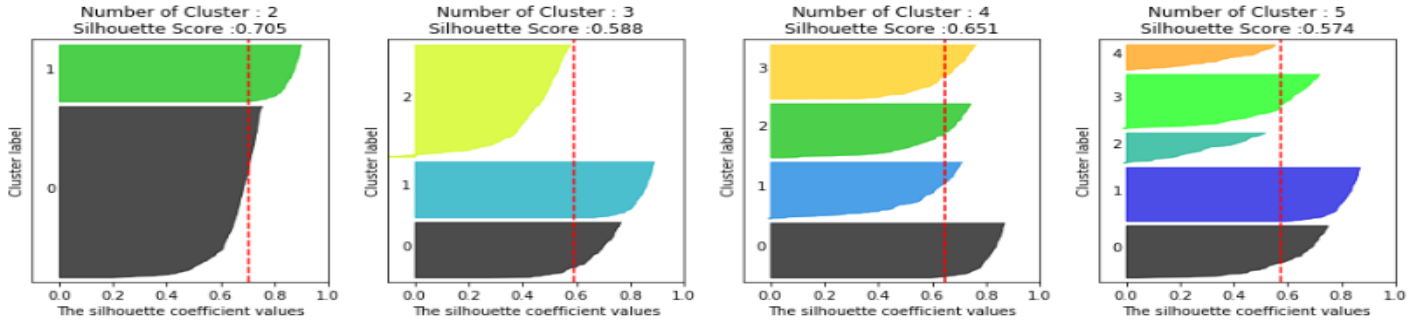
In [7]:
```
irisDF.groupby('cluster')['silhouette_coeff'].mean()
```

Out[7]:
```
cluster
0    0.451105
1    0.798140
2    0.417320
Name: silhouette_coeff, dtype: float64
```

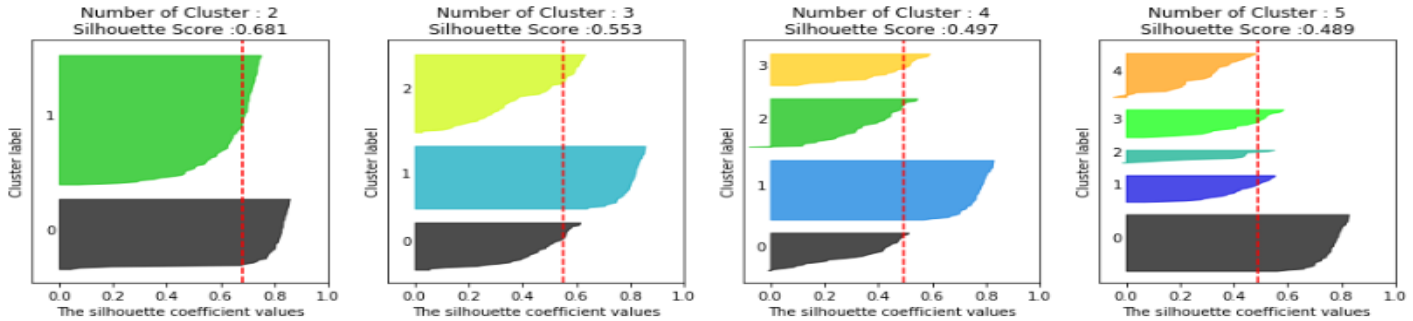
✓ 최적화

✓ 전체 실루엣 계수 + 개별 집단의 실루엣 계수 균등한 분포

```
# make_blobs 를 통해 clustering 을 위한 4개의 클러스터 중심의 500개 2차원 데이터 셋 생성
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=500, n_features=2, centers=4, cluster_std=1, ₩
                  center_box=(-10.0, 10.0), shuffle=True, random_state=1)

# cluster 개수를 2개, 3개, 4개, 5개 일때의 클러스터별 실루엣 계수 평균값을 시각화
visualize_silhouette([ 2, 3, 4, 5], X)
```




```
from sklearn.datasets import load_iris

iris=load_iris()
visualize_silhouette([ 2, 3, 4,5 ], iris.data)
```



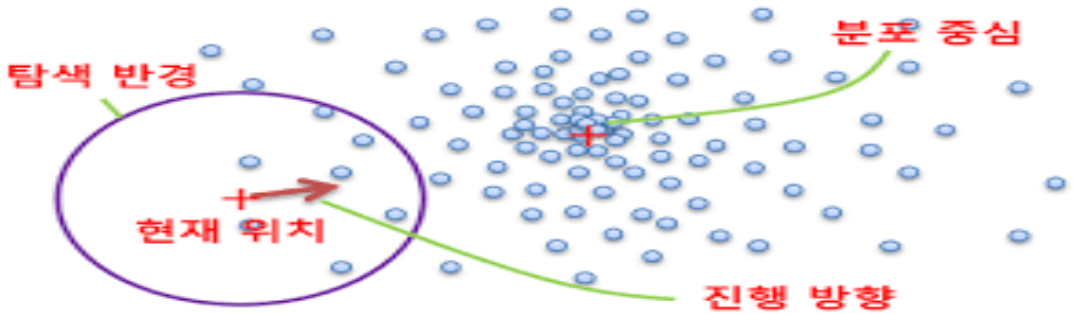
✓ 평균 이동(K-means와 유사한 방법)

- 중심을 데이터가 모여 있는 밀도가 가장 높은 곳으로 이동
- KDE를 이용해 중심을 이동

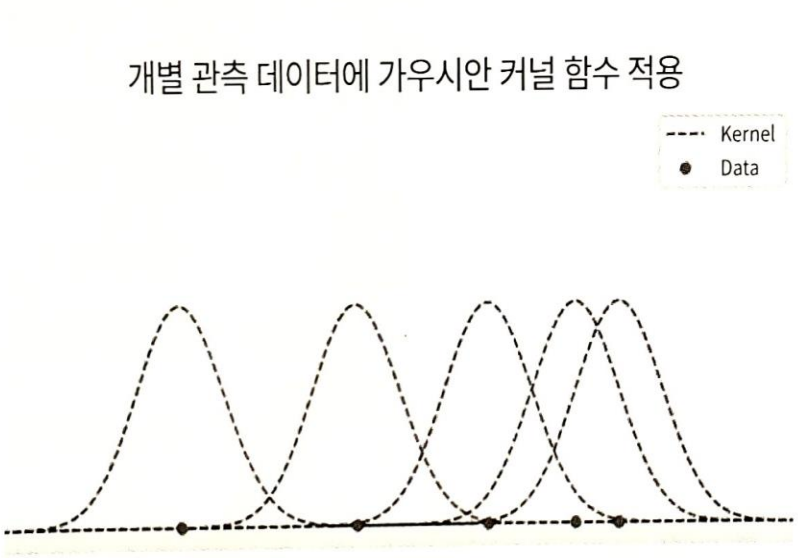
✓ 작동 원리

1. 데이터의 분포도를 KDE 기반의 Mean Shift 알고리즘으로 계산
2. KDE로 계산된 데이터 분포도가 높은 방향으로 데이터 이동
3. 모든 데이터를 1~2까지 수행하면서 데이터를 이동
4. 지정된 반복 횟수만큼 전체 데이터에 대해서 수행
5. 개별 데이터들이 모인 중심점을 군집 중심점으로 설정

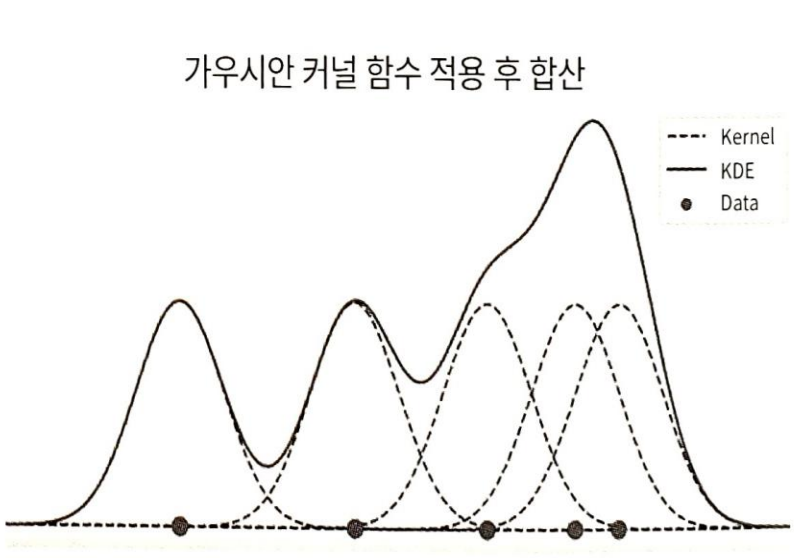
* 중심점을 이동했지만 데이터들의 중심점 소속 변경이 없으면 군집화 완료



✓ 가우시안 분포 함수 사용



개별 관측 데이터에 가우시안 커널 함수를 적용



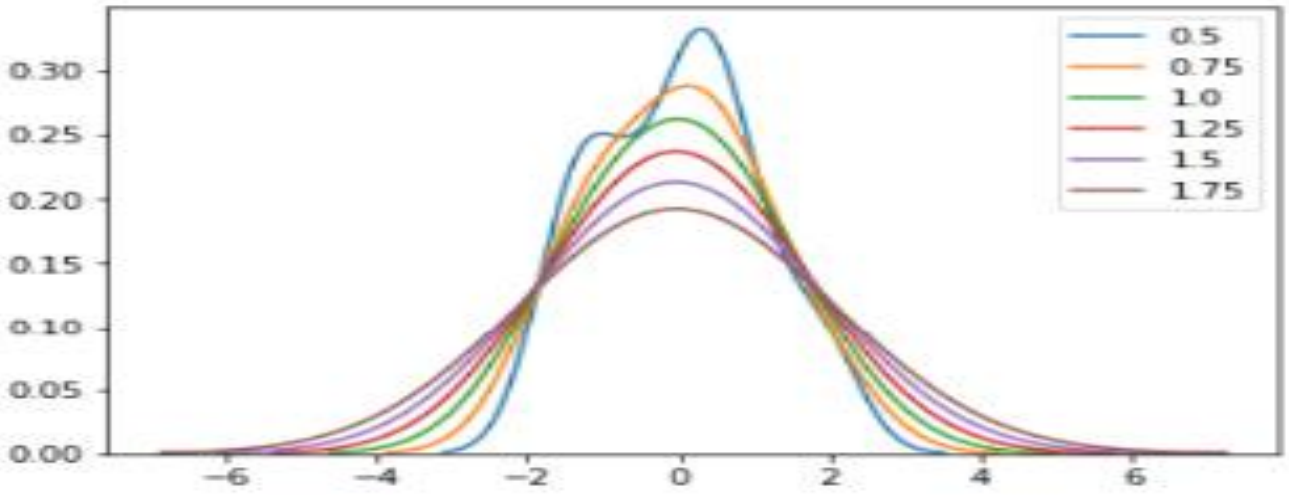
적용 값을 모두 더한 KDE 결과

✓ **KDE**

$$\hat{f}(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i), \; where \; K_h(x) = \frac{1}{h} K\left(\frac{x}{h}\right)$$

✓ **대역폭**(h)에 따른 결과

- ✓ 값이 **작을수록 과적합** 가능성 높음
- ✓ 값이 **클수록 과소적합** 가능성 높음



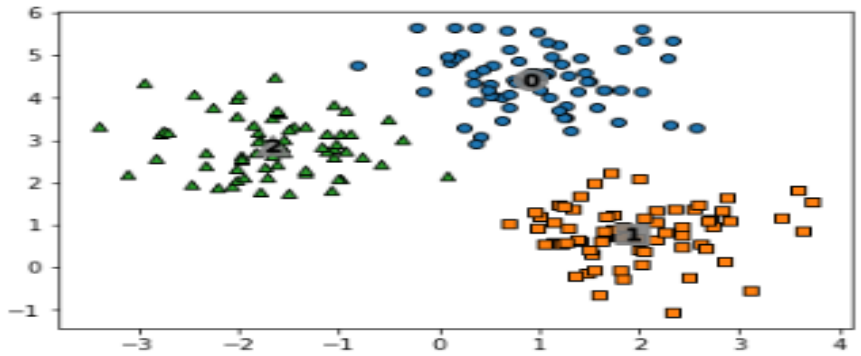
✓ 최적화된 bandwidth 값 찾기

:
```
from sklearn.cluster import estimate_bandwidth

bandwidth = estimate_bandwidth(X)
print('bandwidth 값:', round(bandwidth,3))
```

```
bandwidth 값: 1.816
```

✓ 군집 시각화

```
import matplotlib.pyplot as plt
%matplotlib inline

clusterDF['meanshift_label']  = cluster_labels
centers = meanshift.cluster_centers_
unique_labels = np.unique(cluster_labels)
markers=['o', 's', '^', 'x', '*']

for label in unique_labels:
    label_cluster = clusterDF[clusterDF['meanshift_label']==label]
    center_x_y = centers[label]
    # 군집별로 다른 마커로 산점도 적용
    plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], edgecolor='k', marker=markers[label] )

    # 군집별 중심 표현
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=200, color='gray', alpha=0.9, marker=markers[label])
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', edgecolor='k', marker='$%d$' % label)

plt.show()
```




```
print(clusterDF.groupby('target')['meanshift_label'].value_counts())
```

```
target  meanshift_label
0       0                 67
1       1                 67
2       2                 66
Name: meanshift_label, dtype: int64
```

✓ DBSCAN

- **밀도 기반 군집화**의 대표적인 알고리즘
- **간단**하고 **직관적**이지만 **기하학적 형태**(원 모양)의 데이터 **군집화** 가능

✓ 파라미터

- 입실론 주변 영역 : 개별 데이터를 중심으로 입실론 반경을 갖는 원형의 영역
- 최소 데이터 개수 : 개별 데이터의 입실론 주변 영역에 포함되는 타 데이터의 개수

✓ 데이터 포인트

- **핵심 포인트(Core Point)**: 주변 영역 내에 최소 데이터 개수 이상의 타 데이터를 가지고 있을 경우 해당 데이터를 핵심 포인트라고 합니다.
- **이웃 포인트(Neighbor Point)**: 주변 영역 내에 위치한 타 데이터를 이웃 포인트라고 합니다.
- **경계 포인트(Border Point)**: 주변 영역 내에 최소 데이터 개수 이상의 이웃 포인트를 가지고 있지 않지만 핵심 포인트를 이웃 포인트로 가지고 있는 데이터를 경계 포인트라고 합니다.
- **잡음 포인트(Noise Point)**: 최소 데이터 개수 이상의 이웃 포인트를 가지고 있지 않으며, 핵심 포인트도 이웃 포인트로 가지고 있지 않는 데이터를 잡음 포인트라고 합니다.